Practical Program Understanding With Type Inference

Robert O'Callahan and Daniel Jackson

May 1996
CMU-CS-96-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

19960606 005

Abstract

Many questions that arise in the reverse engineering or restructuring of a program can be answered by determining, statically, where the structure of the program requires sets of variables to share a common representation. With this information we can find abstract data types, detect abstraction violations, identify unused variables, functions, and fields of data structures, detect simple errors of operations on abstract datatypes (such as failure to close after open), and locate sites of possible references to a value.

We have a method for computing representation sharing by using types to encode representations. We use polymorphic type inference to compute new types for all variables, eliminating cases of incidental type sharing where the variables might have different representations. The method is fully automatic and smoothly integrates pointer aliasing and higher-order functions. Because it is fully modular and computationally inexpensive, it should scale to very large systems.

We show how we used of a prototype tool to analyze *Morphin*, a 17,000 line robot control program written in C, answering a user's questions about program structure, detecting abstraction violations, and finding unused data structures and memory leaks.

This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330.

The views and conclusions contained in this document are those of the author[s] and should not be interpreted as representing the official policies, either expressed or implied, of NSF, TRW, Wright Laboratory, ARPA, or the U.S. government.

DTIC QUALITY INSPECTED 1

Keywords: restructuring, reverse engineering, abstraction, C, representation

1 Introduction

Many interesting properties of programs can be described in terms of constraints on the underlying representation of data values. As an obvious example, if a value is an instance of an abstract data type, then the client code must not constrain its representation. If the representations of two different supposedly abstract types are constrained to be the same, there is an abstraction violation. Less obviously, the value of a variable is never used if the program has no constraints on its representation. A necessary condition for a value defined at one site to be used at another is that the two values must have the same representation[1]. By extending the notion of representation (for example, by distinguishing constants from mutables) we can encode other kinds of useful information.

This characterization is profitable because these constraints can be solved using type inference. We can apply type inference to compute new types for the variables and textual expressions of a program, ignoring any actual type declarations and taking account only of the operations performed on the values; the primitive operations of the language induce constraints on the representations of their arguments. We recover a solution to the system of constraints from the inferred types. The type system we use to perform type inference can be very different to the type system for the source language[2].

Type inference is a very attractive implementation technique for many reasons. It is simple and well-understood. It is efficient in practice (our system usually consumes space and time little more than linear in the size of the program being analyzed). It is fully automatic. It generalizes easily to rich source languages, such as languages with recursive pointer-based data structures and function pointers. It is sound for source languages with appropriate semantics[3], in which case the results are guaranteed to be conservative.

We have built a tool ("Lackwit") to demonstrate the feasibility of applying type inference analyses to C programs for program understanding tasks, and to experiment with the kind and quality of information available. The general architecture of the tool is shown in Figure 2. The multiple downward arrows indicate that C modules can be processed individually and fed into the database; the fat upward arrows show where information about the entire system is being passed. In the remaining sections of this paper, we will describe the intermediate code format, the translation process, and the type inference

[1] Some languages may allow a single value to be viewed as having different types at different points in a program, for example by implicit coercions. However these are just views of a single underlying representation; any meaningful transmission of data must use an agreed common representation.

[2] It is very important not to get the two notions of "type" confused. In this paper we will normally be referring to types in our specialized type system.

[3] Arguably, C does not have a well-defined semantics. The behaviour of C's "unsafe" constructs cannot be fully specified in any reasonable way, so any non-trivial static analysis, such as ours, must be unsound.

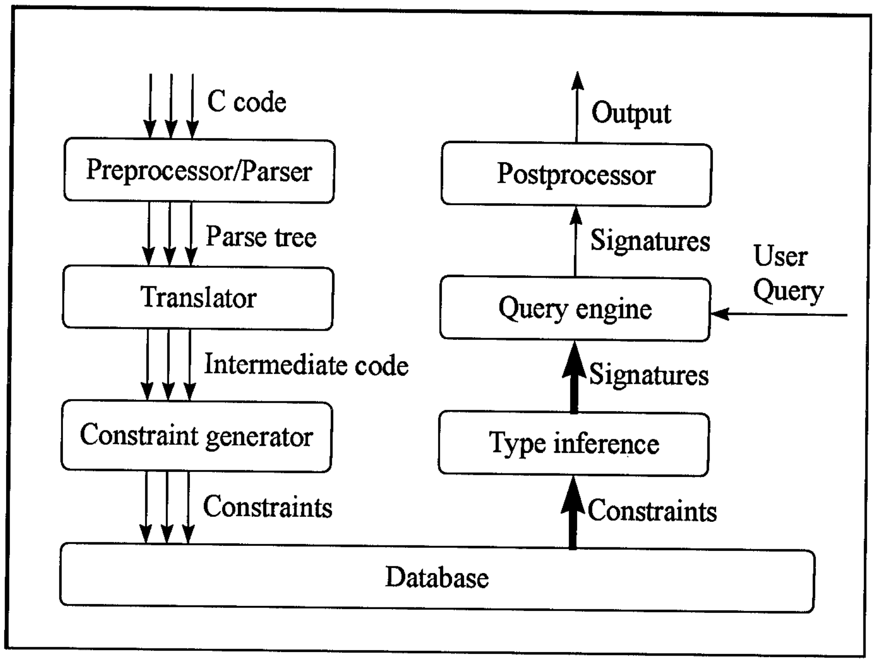

Figure 2: Tool Architecture

system. We will present the results of applying our tool to some real-life C programs and discuss some user interface issues.

Our system has advantages over all the other tools we know of for analyzing source code for program understanding; see Figure 1. Lexical tools [H88, MTOCM92, MN95] lack semantic depth in that they fail to capture effects such as aliasing that are vital for understanding the manipulation of data in large programs. Dataflow-based tools such as the VDG slicer [E94] and Chopshop [JR94] do not scale to handle very large programs. Code checking tools such as LCLint [EGHT94] do not try to present high-level views of a large system. (Furthermore, although LCLint does compute some semantic information, it does not have a framework for accurate global analysis.)

Tool	Semantic depth	System-level display	Scalability
grep	—	—	✓
Rigi, RMT	—	✓	✓
VDG slicer	✓	—	✓
Chopshop	✓	✓	—
LCLint	✓	—	✓
Lackwit	✓	✓	✓

Figure 1: Tool feature comparison

2 Example

Consider the trivial program in Figure 3. In C, the declared type of a variable usually determines its representation, but that is simply a matter of convenience. The program would still operate correctly if we had several different representations of *int*, provided

```
int x;
int p1;

void f(int a, int b, int * c, int * d)
{ x = a;
  *c = *d;
}

void g(int p, int * q, int * r, int * s)
{ int t1 = 2;
  int c1, c2;
  int p;
  p = p1;
  x++;
  f(c1, p, &t1, q);
  f(c2, 4, r, s);
}
```

Figure 3: Trivial program

that each version of *int* had the same semantics and that they were used consistently[4]. Intuitively, we could determine consistency by augmenting the declared type with the representation, and type-checking the program.

Figure 4 shows a consistent augmentation of the trivial program. The names ending in capital letters indicate *polymorphic type variables*: for any choice of representations *intA* and *intB*, the function *f* would type-check; we might just as well have different versions of *f* for each choice of *intA* and *intB*. This means that in *g* we are free to assign *q* and *s* different representations. Note that because of the reference to *x*, the first argument of *f* is not polymorphic; its representation is fixed across different calls to *f*.

```
int1 x;
int2 p1;

void f(int1 a, intA b, intB * c, intB * d)
{ x = a;
  *c = *d;
}

void g(intY * q, intX * r, intX * s)
{ intY t1 = 2;
  int1 c1, c2;
  int2 p;
  p = p1;
  x++;
  f(c1, p, &t1, q);
  f(c2, c2, r, s);
}
```

Figure 4: Trivial program annotated with representations

[4] Of course we do not propose implementing such a scheme. This is merely a pedagogical device.

Suppose that we specify that the arguments of all primitive arithmetic operations have representation *int1*. Then Figure 4 is still consistent. If *p* is intended to be an abstract file descriptor, we can see from its annotated type alone (the fact that it need not be *int1)* that no arithmetic operations can be performed on it; we can conclude that abstraction is preserved.

To reduce the incidental sharing of representations, which lead to spurious warnings, we would like to compute the most general assignment of representations to variables that is consistent with the communication patterns of the program. If we view the representations themselves as types, then this is the problem of type inference [M78].

In the program above, type inference would proceed as follows. Let *x* have type *intZ*, and *p1* have type *intK*. In *f*, because of the assignment of *a* to *x*, *x* and *a* must have the same type, so *a* has type *intZ*. **c* and **d* must have the same type, so let their types be "pointer to *intB*". b is unconstrained so let it have type *intA*. Now we observe that the choices of *intA* and *intB* are arbitrary, and therefore *f* is polymorphic in *intA* and *intB*.

Now, in *g* we perform arithmetic on *x*, so *x* must be type *int1* and we have *intZ* = *int1*. We find that *p* has the same type as *p1*, which is *intK*. In the first call to *f* we instantiate *intA* to be some fresh variable *intW* and *intB* to be some fresh *intY*; we find that *c1* must have type *int1*, *p* is *intW*, and *t1* and *q* are *intY* and "pointer to *intY*" respectively. Since *p* is simultaneously *intK* and *intW*, we must set *intW* = *intK*. In the second call to *f* we instantiate *intA* to be some fresh variable *intV* and *intB* to be some fresh *intX*; then we find that *c2* has type *int1*, *intV* must be *int1*, and *r* and *s* are both "pointer to *intX*". Because *intX* and *intY* were arbitrary choices, the previous derivation is valid for all possible values of *intX* and *intY*, and therefore *g* is polymorphic in all its arguments. Finally, we note that *p1* is of some type *intK* where *intK* is arbitrary but fixed (intuitively, we can't have different versions of *p1* with different representations, because that would destroy sharing). Therefore we set *intK* to be some arbitrary representation *int2*; for maximum generality, we make each such choice unique.

3 Intermediate code format

We have chosen to begin by translating C programs into an intermediate form which we call Q6. Once we have an algorithm to faithfully translate from C to Q6, we only need to consider the analysis of Q6. This makes the analysis somewhat independent of the choice of source language (for example, it would be easy to retarget to analyze Pascal), and also greatly simplifies the presentation.

Programs in Q6 are expressions:

e ::=	x	(Variable reference)
|	**fun**(x) { e }	(Function declaration)
|	$e_1(e_2)$	(Function application)
|	**let** $x_1 = e_1$; … ; $x_n = e_n$ **in** e	(Sequential variable binding)
|	**letrec** $x_1 = e_1$; … ; $x_n = e_n$ **in** e	(Mutually recursive variable binding)
|	**if** e_1 **then** e_2 **else** e_3	(Conditional expression)

Q6 programs are provided an initial environment that includes functions for all the usual primitive operations, as well the operators and unsafe constructs required by C. In particular, we have primitives for memory cell creation, dereference and update, tuple construction and extraction, numeric constants, and arithmetic operators.

4 Translation from C to Q6

The basic ideas behind the translation are simple, though the details are messy. A guiding principle has been to preserve execution semantics when unsafe constructs are not being used, and to do something sensible when they are. This means we actually preserve much more information during translation than we actually need for the analysis (such as information about the execution order of statements), but for research it gives us maximum flexibility to experiment with the analysis. See Figure 5 for an example of the output of our translator. (The translator does little optimization of the Q6 code; here we have eliminated some redundant code for clarity.)

A C program can be viewed as a set of global definitions of variables and functions, possibly mutally recursive. We process each definition individually, translating it into a Q6

```
int f(int * x, int n, int a) {
  int i;
  for (i = 0; i < n; i++)       /* statement 1 */
    if (x[i] == a)              /* statement 2 */
      return i;                 /* statement 3 */
  return -1;                    /* statement 4 */
}

f = fun(args) {
  let a = #1 args
      n = #2 args
      x = #3 args
  in letrec i = ref 0
            stmt1_a = fun(dummy) {
                        stmt1_b( (fun(dummy) { i })(assign(i)(0)) )
                      }
            stmt1_b = fun(dummy) {
                        if <(deref(i))(deref(n)) then stmt2( () )
                          else stmt4( () )
                      }
            stmt1_c = fun(dummy) { stmt1_b(
                        let v = deref(i)
                        in (fun(dummy) { ref(v) })(assign(i)( +(v)(1) ))
                      ) }
            stmt2 = fun(dummy) {
                      if ==(deref( ptr+(deref(x))(deref(i)) ))(deref(a))
                        then stmt_3( () ) else stmt1_c( () )
                    }
            stmt3 = fun(dummy) { i }
            stmt4 = fun(dummy) { ref(-(1)) }
     in stmt1_a( () )
}
```

Figure 5: Translation example

expression (with an associated identifier). We could treat the entire C program as one large **letrec**[5] of these expressions, but in the type system we use that would not permit the use of polymorphism (see below), so we use a graph algorithm to determine the sets of mutually recursive declarations and then order the declarations into a sequence of **letrecs**.

All C variables are mutable[6], so a C variable i is translated to a Q6 variable i bound to a memory location containing the actual value. C structures are converted to Q6 tuples that contain the fields in the order in which they were declared. (Like variables, structure fields are bound to a memory location with their value.) To handle arrays and C's pointer/array dichotomy, we make each Q6 memory location an infinitely growable array; pointers are implemented as a pair consisting of a memory location and an index, and pointer arithmetic updates the index. (Our type inference does not distinguish the elements of arrays, so there is no cost associated with this device.) It is convenient to treat all functions as having one argument, so we pass multiple arguments using the standard convention of bundling them into a tuple.

To translate irregular control constructs within a function, such as *goto*, we translate each statement into a function taking a dummy argument and returning the result of the source function. So normally, if a statement s1 is followed by s2, then s1 will perform some action and then return the result of a call to s2.

C has many unsafe features, but those normally considered errors can be handled by the translation. The use of uninitialized variables is avoided by initializing all variables to default values[7]. Reading and writing beyond array bounds is avoided by making arrays unbounded[8]. We can safely introduce a NULL pointer value into Q6, with the semantics that any primitive operation given a NULL pointer immediately aborts the program.

Other unsafe features are commonly used and we must emulate them in a way that might not agree with the execution of the original program; this causes unsoundness in our analysis. Unions are treated as structures; programs that use unions consistently as variant records will execute correctly after translation, but programs that use unions to perform arbitrary type casts will not. Explicit non-trivial type casts (or implicit casts to pointer-to-void) are translated to a cast primitive; unfortunately the execution semantics of this

[5] For simplicity of translation, we allow **letrec** to bind variables to expressions that are not functions, providing that such an expression contains no references to identifiers bound in the **letrec** (this disallows "**letrec** a = b ; b = a **in** …").

[6] We ignore "const" - since you can take the address of a "const" variable, it's still necessary to represent it with a memory location.

[7] These simple errors can usually be detected by standard techniques, so we don't try to catch them. However, it turns out that our analysis can distinguish the translator's default initialization from source-level initialization, so we can in fact detect many such errors (including uninitialized components of structures, and in the presence of aliases).

[8] These would make execution excrutiatingly inefficient, but that doesn't matter to us.

primitive cannot be specified in any reasonable way. Functions with a variable number of optional arguments are not passed the optional arguments.

5 Type inference for Q6

We use a simple polymorphic type system with recursive types. Monotypes τ and polytypes σ obey the following productions:

σ	::=	τ	(Monomorphic type)
	|	$\forall\alpha.\sigma_1$	(Polymorphic quantification)
τ	::=	α	(Type variable)
	|	$\tau_1 \rightarrow^{\beta} \tau_2$	(Function)
	|	$\tau_1\ \mathbf{ref}^{\beta}$	(Reference (pointer))
	|	$(\tau_1, \tau_2, \ldots, \tau_n)^{\beta}$	(Tuple)
	|	$\mathbf{number}^{\beta}$	(Scalar)

α is a metavariable that ranges over an infinite set of type variables. If the βs are ignored, this is a completely standard polymorphic type system for simple lambda languages[9]. We use the standard inference algorithm W[10] [M78, DM82] to compute the types of all variables of a source program (which initially no type declarations). (The computations described in Section 2 correspond to steps in the execution of W.)

The βs are tags that we use to track the identity of type constructors[11]. A fresh tag is generated whenever a type constructor is introduced in a new constraint. Whenever we find that two types are constrained to be identical, we unify their two type constructors, and their tags are merged; thus the tags partition the set of occurrences of type constructors. For example, if two variables have **ref** constructors with different tags, then they are not aliases[12,13]. If two variables are tuples with the same tag, then they must be structures of the same abstract type (or abstraction violations have occurred). The tags are simple to implement: they can be treated merely as an extra parameter of the type constructor.

[9] We do not use polymorphic recursion; that is, **let** and **letrec** bindings are the only places where we perform polymorphic generalization.

[10] We use a value restriction on polymorphic **lets** to make side-effects safe [W95].

[11] These tags correspond to the region variables of region inference [TT94].

[12] See Steensgaard's work on points-to analysis [S96]. Similarly, comparing the tags on function types with the tags on declared functions gives us an analysis of higher-order control flow.

[13] Actually in the polymorphic type system we must use a more complicated relation than just tag equality; see below.

For example, for the function in Figure 5 we derive the type

$$\mathrm{f} : \forall\alpha.\forall\beta.(\mathbf{number}^{\alpha}\ \mathbf{ref}\ \mathbf{ref},\ \mathbf{number}^{\beta}\ \mathbf{ref},\ \mathbf{number}^{\alpha}\ \mathbf{ref}) \rightarrow \mathbf{number}^{\beta}\ \mathbf{ref}$$

(tags on **refs** elided for clarity). This expresses the constraint that the numbers in the array and the number being searched for are constrained to have the same representation, and that the array size and the returned array index also have the same representation. We know that the array contents and indices are all integers, but there is more to representation than that; for example, the array values may be in dollar units, but the indices are cardinals. The tags let us distinguish these different encodings. Note that the tags are polymorphic; the representations could be different at each call site. Thus we obtain a context-sensitive analysis of function calls.

We supply type signatures for the built-in primitives and any library functions that are called; these signatures are the only information about external code that we require. Since such signatures are also the result of the analysis, this makes our techniques modular (and contributes to scalability). By adjusting the signatures, we can customize the analysis to compute different kinds of information (see Section 9).

The basic signatures are listed in Appendix 1. The only interesting signature is for the cast operator. If we wish to compute consistent types for all compilable C programs, we must allow a cast to magically convert a value from any type to any other, making the type system unsound[14]. Alternatively, we can treat "cast" as the identity function, in the hope that casts are merely being used to work around the lack of polymorphism in C's type system (and so the program will still be typable in our type system). Even if this is not true, we can still report any type errors, display their context, and continue the analysis. The user may be able to check whether the results they are interested in have been compromised. In our Morphin example below (17,000 lines), there are just two "bad" type casts.

Recursive types are treated as infinite regular trees (see Cardone and Coppo [CC92] for details).

6 Displaying Data

The result of the type inference phase is a mapping from source variables and functions to type signatures in our extended type system. We provide ways to filter out interesting information and display it graphically in a way that has a clear relationship to the source program.

Our basic approach is to produce a graph summarizing the information about a single component of a variable (see Figure 6 below). The nodes of the graph represent top-level declarations, and the edges represent the use of one declaration by another in the text of the program. Arrows point from the using declarations to the used declarations. When we can prove, by inspecting signatures (see below), that a use cannot transmit the value of the

[14] If we do this, then the typing of a C program given by a C compiler can be translated into a valid typing of the corresponding Q6 code, and the type inference algorithm will find a valid typing if one exists, so we will successfully type all compilable C programs.

queried component, then we omit the corresponding edge. By eliminating unreachable nodes, we show just the part of the program that is able to access the value. A value is transmitted by passing a data structure containing it as a parameter in a function call, or by returning such a data structure from a function call, or by referencing global data (containment includes reachability through pointers).

The inspection of signatures used to filter the edges works as follows. The user has specified a global variable, function result, function parameter or local variable, or some component thereof (a chain of pointer dereferences and/or structure fields). Because any accesses to run-time values of the component must agree on the representation, the types of the components through which accesses are made must have tags that are "compatible". with the tag on the type of the queried component. In a monomorphic type system, the "compatibility" relation is just tag equality, but with polymorphism we have to consider that a polymorphic tag may be instantiated to some other tag, which means the same run-time value could be accessible through different tags. For example, in Figure 4, the value of *s* is accessed with tags *intB* and *intX*. Therefore we define two tags *S* and *T* to be "compatible" if there is some tag *U* such that the program exhibits a chain of instantiations from *S* to *U* and a chain of instantiations from *T* to *U* (the chains may be empty). Note that this relation is symmetric but not transitive — in Figure 4 *intB* is compatible with *intX* and *intY*, but *intX* is not compatible with *intY*.

We determine the set of "tags of interest" that are compatible with the tag of the queried component, and locate the declarations whose signatures contain occurrences of tags of interest. Declarations whose signatures do not contain any occurrences of tags of interest do not transmit any interesting values when they are used, so those uses are omitted from the graph. Since polymorphic signatures can be instantiated to different types at different usage sites, we also omit uses when the type at the usage site does not contain any tags of interest.

There is a difficult tradeoff between detail and clarity in choosing what kinds of information to display for each node and edge. We have found it useful to visually distinguish functions from global data, and to highlight nodes that are "interesting" (for example, functions that directly access the representation of some variable). Clearly it would be beneficial to have an interactive display, which we plan to add in future work.

To compute local information (for example, to determine whether or not a function's body constrains the representation of some piece of data), we simply apply the usual type inference to a single declaration. Any declarations that it references are treated as not inducing any constraints. For example, in Figure 6, we determine that the definition of *map_mgr_convert_pixel_coords* does not directly access the representation of the vehicle object, even though it calls a function which does.

7 Results (1)

We have used our tool to analyze Morphin, a robot vehicle control program consisting of over 17,000 lines of C code, with 252 functions and 73 global variables. Morphin is to be restructured and adapted to support new features. The developer responsible for this work asked us to determine where and how certain structures were used, if at all. In the course of answering his questions, Lackwit also highlighted some representation exposures.

We computed results for queries of the form "Which functions in the program could directly access the representation of component X of variable Y?" Figure 6 shows the results of a query on the "current vehicle" field of the *map_manager_global* global variable (we also allow queries involving local variables and function parameters). Given that the "veh_" functions are operations on the vehicle abstract data type, it's easy to see that abstraction may be violated in the functions *map_mgr_process_image*, *map_mgr_comp_range_window*, and *map_mgr_process_geometry_range_window*, but nowhere else.

The shaded nodes are the definitions that directly access representations; that is, whose code constrains the representation of the value in question. In this case, the value in question is a structure, and the shaded nodes constrain the type by accessing fields of the structure.

To give some idea of the performance of our prototype, Lackwit built the constraint database from the 17,000 lines of source in 274 seconds (wall-clock time). The database is about 15MB. The type inference procedure took 78 seconds to solve the constraints, about 23 seconds of which was user-level CPU time. After solving the constraints, individual queries are answered almost instantly. These numbers are for a 90Mhz Pentium with 32MB of RAM running Windows NT. As we will discuss in Section 10, certain optimizations could dramatically improve performance. It should be noted that the process of building the database can be carried out independently for each source module; thus the potential performance bottlenecks are all in the solver.

The type inference algorithm processes functions one at a time, iteratively computing a signature for each function and adding it to a "type environment". While processing a function, the main operation is unification of types, the cost of which is proportional to the smaller of the sizes of the descriptions of the types[15]. The number of unifications performed is proportional to the size of the code for the body of the function, which is preserved (to within a constant factor) by the translator. Therefore, if the size of the inferred types is bounded, the solver takes time and space little more than linear in the size of the program. The types are small in practice.

[15] Actually, because we must maintain equivalence classes of type variables using union-find with path compression, there is a superlinear component of $\alpha(n)$, where α is the inverse Ackermann function. For all practical purposes, $\alpha(n) < 6$.

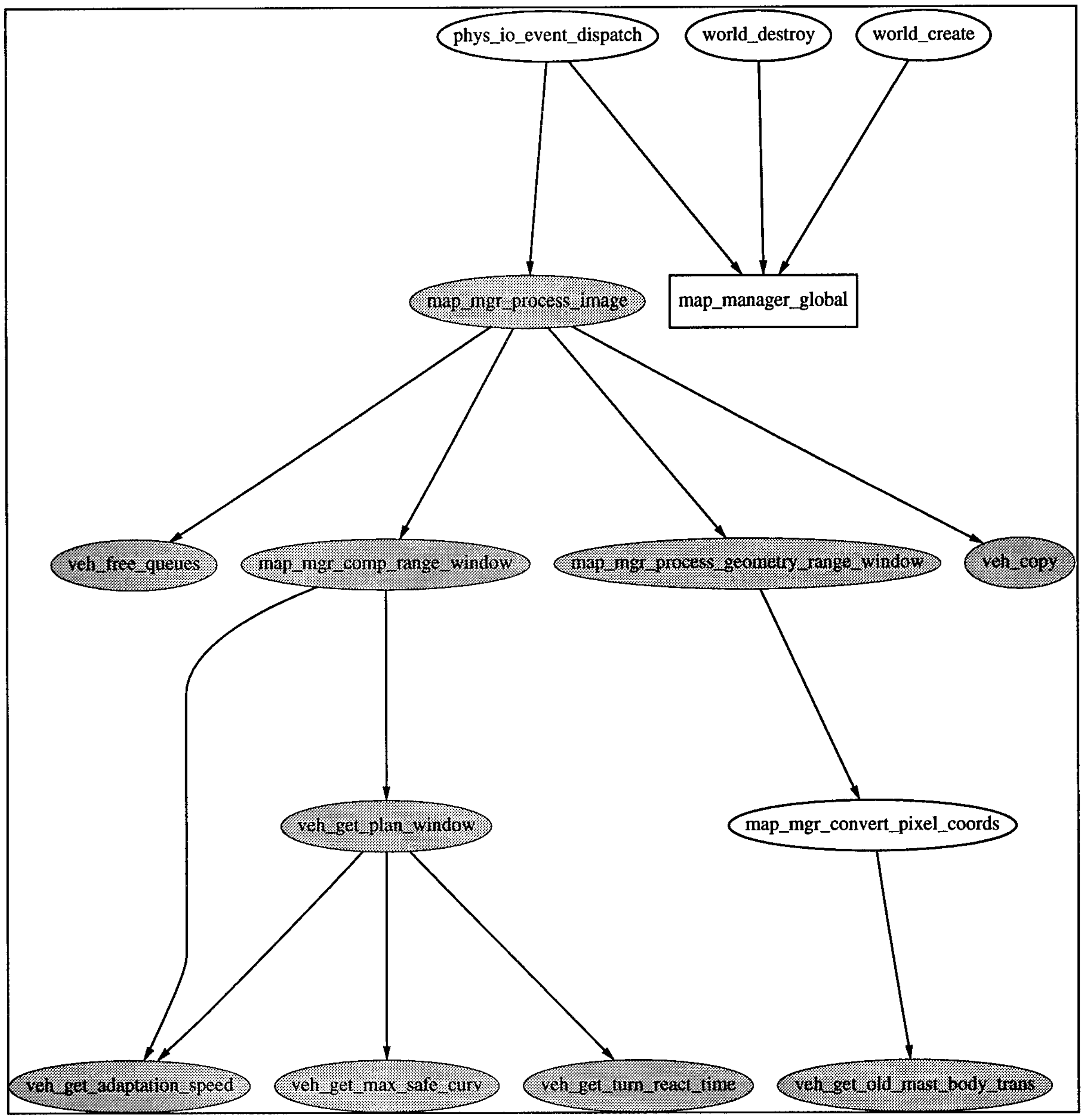

Figure 6: *(map_manager_global->cur_veh)

To attempt to characterize the sizes of query results, we computed for every possible query an upper bound approximation of the number of nodes of the graph we would produce (see Figure 7). Figure 8, Figure 9, and Figure 10 show the sizes of results of queries on all pointers, structures and scalars respectively. The large spike at the right-hand side of Figure 7 and Figure 10 is due to a set of integer variables that are grouped together by chains of arithmetic operations; queries that hit this set (about 65% of all queries on integer-valued components) will probably produce too much information to be useful. However, queries on structures and pointers will produce results of managable size.

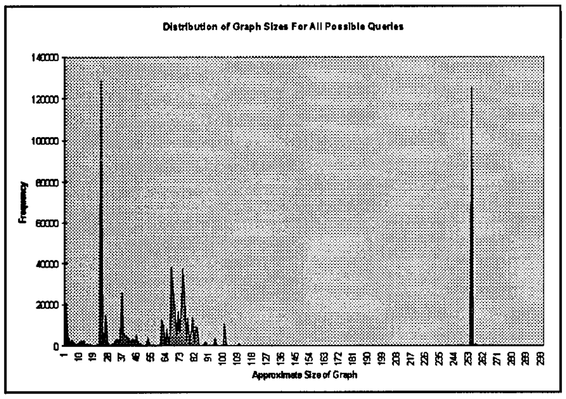

Figure 7: Distribution of Graph Sizes For All Queries

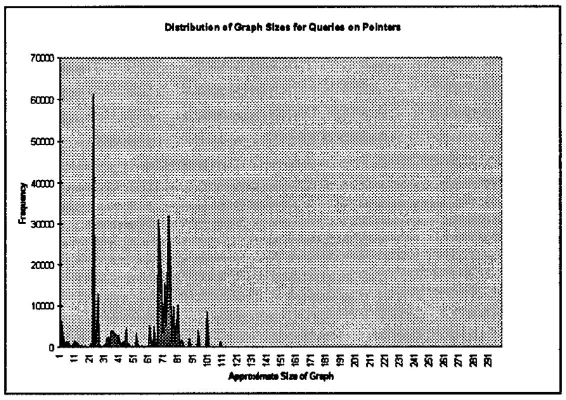

Figure 8: Distribution of Graph Sizes For Queries on Pointers

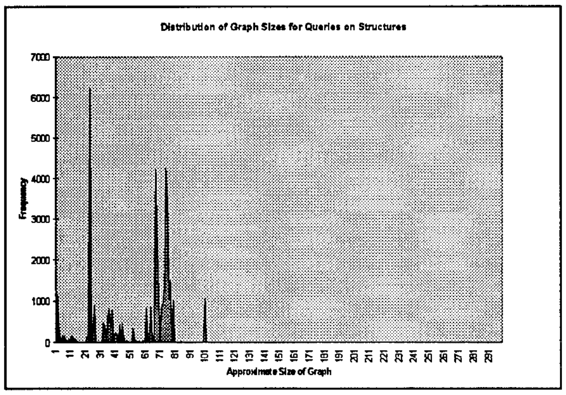

Figure 9: Distribution of Graph Sizes For Queries on Structures

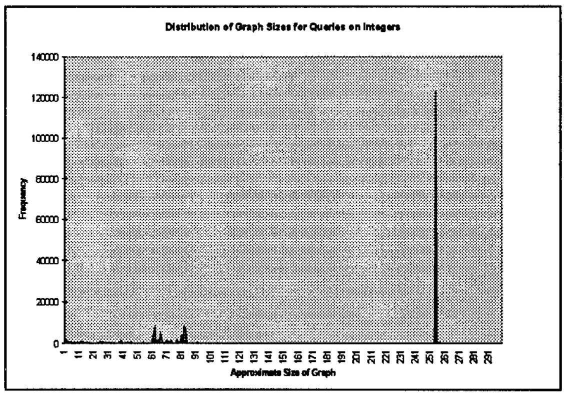

Figure 10: Distribution of Graph Sizes For Queries on Integers

8 Extensions to the type system

We can analyze a broader range of program properties by extending the type sytem in simple ways. For example, it is useful to know whether a memory location is never read or never written, or whether a piece of dynamically-allocated storage is never allocated or never deallocated, or where these effects might occur. We encode such properties by introducing specialized type constructors and a subtype relation (see Figure 11). Primitives such as "assign" and "deref" now constrain their arguments to be "written" refs and "read" refs respectively.

$$\begin{tikzcd}
 & \text{ref} \arrow[dl] \arrow[dr] & \\
\text{ref (read)} \arrow[dr] & & \text{ref (written)} \arrow[dl] \\
 & \text{ref (read and written)} &
\end{tikzcd}$$

Figure 11: Read/write attributes for memory locations (arrows point from supertypes to subtypes)

This is easy to implement when the properties are a vector of boolean values and all constraints are of the form "property p is true"[16]. We simply use our existing type inference procedure and attach additional parameters to the type constructors, one for each boolean. Operations that constrain a property of their arguments or results use the special type *unit* as the appropriate parameter of the type constructor, otherwise they have a type variable. For example, if we give a pointer to α the type "(α, *read*, *written*) ref", then the signature of "deref" becomes "$\forall\alpha.\forall\beta.(\alpha, \textit{unit}, \beta)$ ref $\rightarrow \alpha$".

When type inference is finished and we have the signatures of all the functions and local and global variables, we can easily inspect them to discover anomalous types, such as memory locations that are never read (the "read" parameter is a type variable), or dynamic storage that may be allocated but is never deallocated (the "allocated" parameter is *unit* but the "deallocated" parameter is a type variable).

If the condition of interest can be written in the form "P and not Q" (this includes all the examples we have presented), then we do not report an anomaly unless one actually exists (assuming that there are no errors introduced by the translation from C to Q6, and that there is no "dead code" – for every call to a primitive operation in the program, there is some input that will cause it to be executed). For if a type has property P, then there must be some call to a primitive operation that constrains the type to have that property, and since there is no dead code, there is some execution that invokes this operation (so there will be a run-time value V with property P). If the type does not have property Q, then there is no operation that constrains the type to be Q, and so the run-time value V cannot obtain property Q.

Although we do not report spurious anomalies, we can only give an upper bound approximation to the actual site(s) of any problems, based on the occurrences of the type tag as described in Section 6.

[16] Although "all properties are true" will always be a solution, we will recover the most general solution. For example, we may discover that a program is consistent with a memory location being either "read" or "unread", in which case we may legitimately treat it as "unread" (and eliminate the offending variable or structure element).

9 Results (2)

We used these techniques to perform two specialized analyses: detection of data that is never read, and detection of memory leaks. These correspond to checking, for each type that is a memory location, whether the location is created (by a variable entering scope or by dynamic allocation) and not read, or whether the location is dynamically allocated and not dynamically deallocated. All reported candidates were shown to be non-spurious by human inspection of the source code.

Lackwit reported nine global variables that are never read and ten local variables that are never read. In addition, it reported that six local variables are structures containing some fields that are never read.

Checking for memory leaks, Lackwit reported that six global variables refer to dynamic data structures that are never freed. (These do not cause problems in practice, since they are freed by the operating system when the process terminates, but they are poor programming style.) It also discovered two fields of data structures that are pointers to memory that is never freed. These are a genuine problem, because these pointers are updated in a frequently-executed loop.

10 Current Status and Future Work

The Lackwit front end is currently written in C and C++, and is based on the PCCTS toolkit [PDC92]. The database is simply a sequential binary file, implemented by hand in C. The solver and query engine are also written in C; currently the query language is very simple and a bit unwieldy. The query engine outputs relational tables in a text format. A Perl script converts these tables to a graph, in the process performing the postprocessing analyses described above. We use the “dot” graph-drawing tool [GNV88] to produce the graph.

Performance is currently good, but could be greatly improved. In particular, the recursive-descent parser should be rewritten as an LALR parser for speed. We would gain a lot of performance at the cost of flexibility and simplicity by eliminating Q6 and generating constraints directly from the C abstract syntax (as Steensgaard does [S96]). Most importantly, simplifying constraints in the front end would produce at least an order of magnitude saving in the size of the database and the processing time of the solver. This is important because the solver is the potential performance bottleneck as the number of source files increases.

To make the tool easy to use, we need a better query interface, preferably providing the program source code as context. The tables output by the query engine can be very large, suggesting that most of the postprocessing should be folded into the query engine (to reduce output and parse time). Perhaps our biggest problem is the graph-drawing program, “dot”. It does not scale well to large, highly-connected systems. We would like to explore the use of interactive visualizations or other techniques to present the data in a more useful way.

We intend to experiment with alternative type systems. We may be able to incorporate recent work on type inference of ill-behaved programs. We would like to encode more information in types to increase the scope and accuracy of these techniques. Another interesting problem is to enrich the type sytem to handle a wider class of source languages, for example by adding subtyping for object-oriented languages.

11 Related Work

Our basic analysis technique is similar to "region inference", used by Tofte and Taplin [TT94] to improve the space efficiency of implementations of functional languages. The store is partitioned into distinct "regions", and each value is associated with a region, in the same way that we associate values with representation types; however, we have no analogue to their approximation of the side effects of functions. To our knowledge, we are the first to use these techniques for program understanding.

Other researchers have been investigating type inference methods for inferring properties of C programs. In [S96] Steensgaard presents a method based on type inference that yields an almost-linear time "points-to" analysis. That algorithm is monomorphic (context-insensitive) and does not distinguish elements of compound structures, but variants have been constructed that overcome these limitations.

Bowdidge and Griswold's "star diagram" tool aids in encapsulating abstract data types [BG94]. They assume that there is a single global variable to be abstracted, but they discuss extending their method to operate on data structures with multiple instances. They consider operating on all data structures of a certain type, but comment "The potential shortcoming of this approach is that two data structures of the same representation type, particularly two arrays, might be used for sufficiently different purposes that they are not really instances of the same type abstraction". Our method provides an answer to this problem.

Muller et al. [MTOCM92] have proposed a reverse engineering technique in which first a static analysis is performed, and then the graphical output is visualized and manipulated by the user with the help of various automatic tools, to reveal and impose structure. Our analysis is more powerful than that incorporated in their Rigi tool, but we would certainly benefit greatly from such visualization and manipulation techniques.

LCLint [EGHT94] is a tool that finds inconsistencies between C programs and simple specifications. There is some overlap between the properties they are able to check and ours (for example, some abstraction violations and unused data), but their methods cannot simultaneously distinguish different instances of the same C type and handle complex data structures. On the other hand, their checks incorporate more information, such as flow-sensitive dataflow analysis, so they will catch many errors that we cannot. The two tools are complementary.

12 Acknowledgements

We would like to thank Robert Harper and Lars Birkedal for their helpful advice. We are very grateful to Reid Simmons for taking the time to explain his needs to us, and for allowing us to use his code. We would also like to thank the members of the CMU Software Group for their help and feedback.

13 References

[BG94] Robert W. Bowdidge and William G. Griswold. Automated support for encapsulating abstract data types. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans, December 1994.

[CC92] F. Cardone and M. Coppo. Type inference with recursive types: syntax and semantics. *Information and Computation*, 1992, number 1, pp. 48-80.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. *Proceedings of the Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1982, pp. 207-212.

[E94] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical report MSR-TR-94-14, Microsoft Research, Microsoft Corporation, Redmond, July 1994.

[EGHT94] D. Evans, J. Guttag, J. Horning, and Y. Tan. LCLint: a tool for using specifications to check code. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans, December 1994.

[GNV88] E. Gansner, S. North and K. Vo. DAG — a graph drawing program. *Software Practice and Experience*, volume 18, number 11, November 1988, pp. 1055-1063.

[H88] A. Hume. A tale of two greps. *Software Practice and Experience*, volume 18, number 11, November 1988, pp. 1063-1072.

[JR94] Daniel Jackson and Eugene Rollins. Abstractions of program dependences for reverse engineering. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans, December 1994.

[M78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348-375, 1978.

[MN95] Gail Murphy and David Notkin. Lightweight source model extraction. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, 1995.

[MTOCM92] H. Müller, S. Tilley, M. Orgun, B. Corrie and N. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In SIGSOFT '92*: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Devleopment Environments* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88-98, December 1992. In *ACM Software Engineering Notes*, 17(5).

[PDC92] T. Parr, H. Dietz, and W. Cohen. PCCTS Reference Manual (version 1.00). *ACM SIGPLAN Notices*, February 1992, pp. 88-165.

[S96] Bjarne Steensgaard. Points-to analysis in almost linear time. *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, January 1996 (draft version sighted).

[TT94] Mads Tofte and Jean-Pierre Taplin. Implementation of the typed call-by-value λ-calculus using a stack of regions. *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994, pp. 188-201.

[W95] Andrew Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, volume 8, number 4, December 1995, pp. 343-356.

Appendix 1: Built-in signatures

The following primitives are used by the translator.

Primitive	**Signature**
ref	$t \to t\ \mathbf{ref}^{\alpha}$
assign	$t\ \mathbf{ref}^{\alpha} \to t \to ()$
deref	$t\ \mathbf{ref}^{\alpha} \to t$
mk-tuple$_n$	$t_1 \to t_2 \to \ldots\ t_n \to (t_1, t_2, \ldots, t_n)^{\alpha}$
elem-tuple$_{n,k}$	$(t_1, t_2, \ldots, t_n)^{\alpha} \to t_k$
ref-array$_n$[17]	$(t, t, \ldots, t)^{\alpha} \to t\ \mathbf{ref}^{\beta}$
copy-array[18]	$(t\ \mathbf{ref}^{\alpha} \to t) \to t\ \mathbf{ref}^{\alpha} \to t\ \mathbf{ref}^{\beta}$
undefined-scalar[19]	t
NULL	$t\ \mathbf{ref}^{\alpha}$
cast	$t \to u$ *OR* $t \to t$
scalar$_n$	$\mathbf{number}^{\alpha}$
pointer-arith$_{op}$	$t\ \mathbf{ref}^{\beta} \to \mathbf{number}^{\alpha} \to t\ \mathbf{ref}^{\beta}$
unary-arith$_{op}$	$\mathbf{number}^{\alpha} \to \mathbf{number}^{\alpha}$
binary-arith$_{op}$	$\mathbf{number}^{\alpha} \to \mathbf{number}^{\alpha} \to \mathbf{number}^{\alpha}$
binary-relational$_{op}$	$\mathbf{number}^{\alpha} \to \mathbf{number}^{\alpha} \to \mathbf{number}^{\beta}$
pointer-relational$_{op}$	$t\ \mathbf{ref}^{\beta} \to t\ \mathbf{ref}^{\beta} \to \mathbf{number}^{\alpha}$
pointer-unary$_{op}$	$t\ \mathbf{ref}^{\beta} \to \mathbf{number}^{\alpha}$

Pointer arithmetic operators: pointer add, pointer subtract

Unary arithmetic operators: negate, bitwise not, logical not, unary plus

Binary arithmetic operators: add, subtract, multiply, divide, modulus, left shift, right shift, bitwise and, bitwise or, bitwise xor

Binary relational operators: less than, greater than, equal, less than or equal, greater than or equal, not equal

Pointer relational operators: less than, greater than, equal, less than or equal, greater than or equal, not equal

Pointer unary operators: logical not

[17] This is used to translate array initializers.

[18] This is used when we assign a structure value that contains an array.

[19] This is used to initialize scalars that aren't initialized in C. This allows us to store pointers in them, allowing us to handle programs that use integers polymorphically as integers or pointers.